5 Mapping URLs to Razor Pages using routing

This chapter covers

- Mapping URLs to Razor Pages
- Using constraints and default values to match URLs
- Generating URLs from route parameters

In chapter 4 you learned about the MVC design pattern and how ASP.NET Core uses it to generate the UI for an application using Razor Pages. Razor Pages contain page handlers that act as mini controllers for a request. The page handler calls the application model to retrieve or save data. The handler then passes data from the application model to the Razor view, which generates an HTML response.

Although not part of the MVC design pattern per se, one crucial part of Razor Pages is selecting which Razor Page to invoke in response to a given request. This process is called *routing* and is the focus of this chapter.

This chapter begins by identifying the need for routing and why it's useful. You'll learn about the endpoint routing system introduced in ASP.NET Core 3.0, see several examples of routing techniques, and explore the separation routing can bring between the layout of your Razor Page files and the URLs you expose.

The bulk of this chapter focuses on how to use routing with Razor Pages to create dynamic URLs, so that a single Razor Page can handle requests to multiple

URLs. I'll show you how to build powerful route templates and give you a taste of the available options.

In section 5.5, I describe how to use the routing system to *generate* URLs, which you can use to create links and redirect requests for your application. One of the benefits of using a routing system is that it decouples your Razor Pages from the underlying URLs that are used to execute them. You can use URL generation to avoid littering your code with hardcoded URLs like `/Product/View/3`. Instead, you can generate the URLs at runtime, based on the routing system. The benefit of this is that it makes changing the URL configuration for a Razor Page easier. Instead of having to hunt down everywhere you used the Razor Page's URL, the URLs will be automatically updated for you, with no other changes required.

I finish the chapter by describing how you can customize the conventions Razor Pages uses, giving you complete control over the URLs your application uses. You'll see how to change the built-in conventions, such as using lowercase for your URLs, as well as how to write your own convention and apply it globally to your application.

By the end of this chapter, you should have a much clearer understanding of how an ASP.NET Core application works. You can think of routing as the glue that ties the middleware pipeline to Razor Pages and the MVC framework. With middleware, Razor Pages, and routing under your belt, you'll be writing web apps in no time!

5.1 What is routing?

Routing is the process of mapping an incoming request to a method that will handle it. You can use routing to control the URLs you expose in your application. You can also use routing to enable powerful features like mapping multiple URLs to the same Razor Page and automatically extracting data from a request's URL.

In chapter 3 you saw that an ASP.NET Core application contains a middleware pipeline, which defines the behavior of your application. Middleware is well suited to handling both cross-cutting concerns, such as logging and error handling, and narrowly focused requests, such as requests for images and CSS files.

To handle more complex application logic, you'll typically use the `EndpointMiddleware` at the end of your middleware pipeline, as you saw in chapter 4. This middleware can handle an appropriate request by invoking a method, known as a page handler on a Razor Page or an action method on an MVC controller, and using the result to generate a response.

One aspect that I glossed over in chapter 4 was *how* to select which Razor Page or action method to execute when you receive a request. What makes a request "appropriate" for a given Razor Page handler? The process of mapping a request to a handler is called *routing*.

DEFINITION *Routing* in ASP.NET Core is the process of mapping an incoming HTTP request to a specific handler. In Razor Pages, the handler is a page handler method in a Razor Page. In MVC, the handler is an action method in a controller.

At this point you've already seen several simple applications built with Razor Pages in previous chapters, so you've seen routing in action, even if you didn't realize it at the time. Even a simple URL path—for example, `/Index`—uses routing to determine that the Index.cshtml Razor Page should be executed, as shown in figure 5.1.

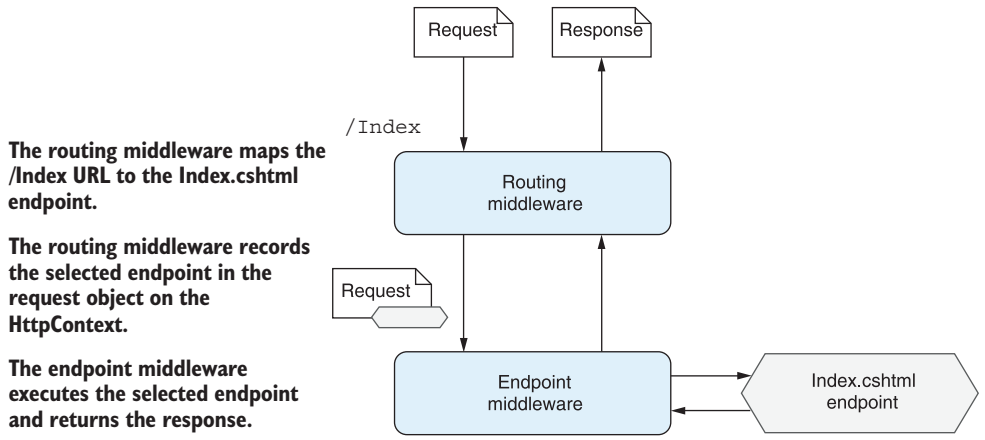

Figure 5.1 The router compares the request URL against a list of configured route templates to determine which action method to execute.

On the face of it, that seems pretty simple. You may be wondering why I need a whole chapter to explain that obvious mapping. The simplicity of the mapping in this case belies how powerful routing can be. If this file layout-based approach were the only one available, you'd be severely limited in the applications you could feasibly build.

For example, consider an e-commerce application that sells multiple products. Each product needs to have its own URL, so if you were using a purely file layout-based routing system, you'd only have two options:

- *Use a different Razor Page for every product in your product range.* That would be completely unfeasible for almost *any* realistically sized product range.
- *Use a single Razor Page and use the query string to differentiate between products.* This is much more practical, but you would end up with somewhat ugly URLs, like `"/product?name=big-widget"`, or `"/product?id=12"`.

DEFINITION The *query string* is part of a URL that contains additional data that doesn't fit in the path. It isn't used by the routing infrastructure for identifying which action to execute, but it can be used for model binding, as you'll see in chapter 6.

With routing, you can have a *single* Razor Page that can handle *multiple* URLs, without having to resort to ugly query strings. From the point of the view of the Razor Page,

the query string and routing approaches are very similar—the Razor Page dynamically displays the results for the correct product as appropriate. The difference is that with routing, you can completely customize the URLs, as shown in figure 5.2. This gives you much more flexibility and can be important in real-life applications for search engine optimization (SEO) reasons.[1]

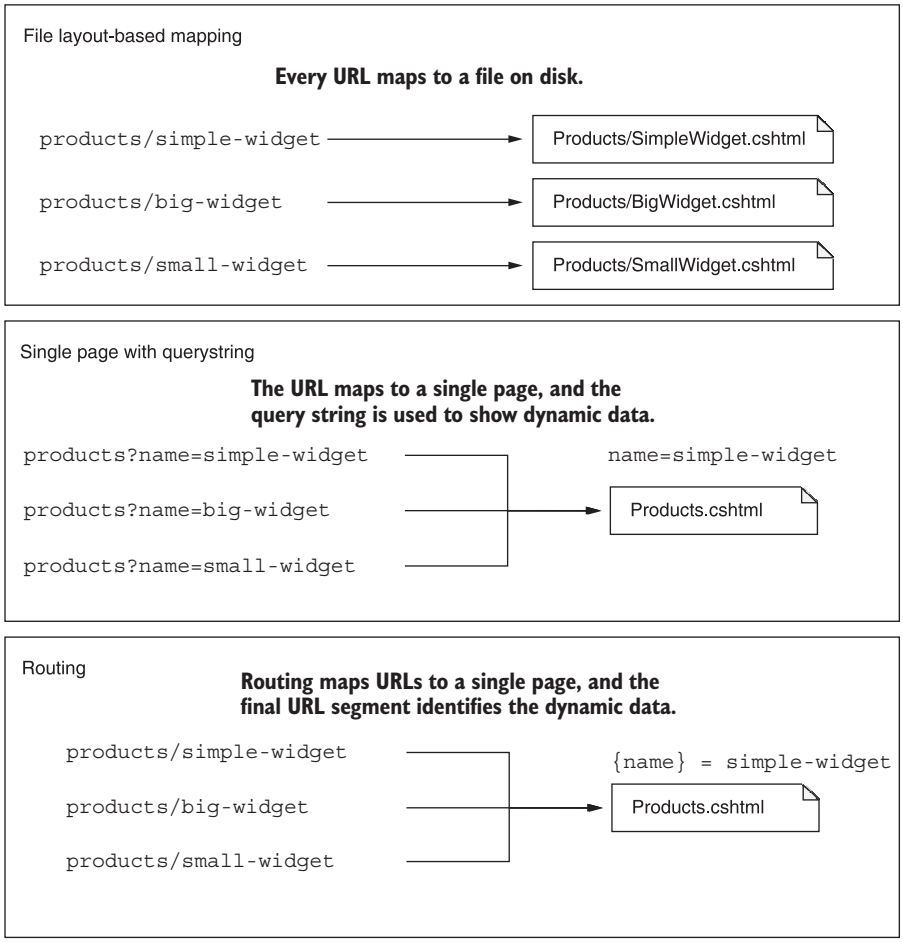

Figure 5.2 If you use file layout-based mapping, you need a different Razor Page for every product in your product range. With routing, multiple URLs map to a single Razor Page, and a dynamic parameter captures the difference in the URL.

[1] Importantly, you can properly encode the hierarchy of your site in your URLs, as described in Google's SEO starter guide: https://support.google.com/webmasters/answer/7451184.

As well as enabling dynamic URLs, routing fundamentally decouples the URLs in your application from the filenames of your Razor Pages. For example, say you had a currency converter application with a Razor Page in your project located at the path `Pages/Rates/View.cshtml`, which is used to view the exchange rate for a currency, say USD. By default, this might correspond to the `/rates/view/1` URL for users. This would work fine, but it doesn't tell users much—which currency will this show? Will it be a historical view or the current rate?

Luckily, with routing it's easy to modify your exposed URLs without having to change your Razor Page filenames or locations. Depending on your routing configuration, you could set the URL pointing to the View.cshtml Razor Page to any of the following:

- `/rates/view/1`
- `/rates/view/USD`
- `/rates/current-exchange-rate/USD`
- `/current-exchange-rate-for-USD`

I know which of these I'd most like to see in the URL bar of my browser, and which I'd be most likely to click! This level of customization isn't often necessary, and the default URLs are normally the best option in the long run, but it's very useful to have the capability to customize the URLs when you need it.

In the next section we'll look at how routing works in practice in ASP.NET Core.

5.2 Routing in ASP.NET Core

Routing has been a part of ASP.NET Core since its inception, but in ASP.NET Core 3.0 it went through some big changes. In ASP.NET Core 2.0 and 2.1, routing was restricted to Razor Pages and the ASP.NET Core MVC framework. There was no dedicated routing middleware in your middleware pipeline—routing happened only within Razor Pages or MVC components.

Given that most of the logic of your application is implemented in Razor Pages, only using routing for Razor Pages was fine for the most part. Unfortunately, restricting routing to the MVC infrastructure made some things a bit messy. It meant some cross-cutting concerns, like authorization, were restricted to the MVC infrastructure and were hard to use from other middleware in your application. That restriction caused inevitable duplication, which wasn't ideal.

In ASP.NET Core 3.0, a new routing system was introduced, *endpoint routing*. Endpoint routing makes the routing system a more fundamental feature of ASP.NET Core and no longer ties it to the MVC infrastructure. Razor Pages and MVC still rely on endpoint routing, but now other middleware can use it too. ASP.NET Core 5.0 uses the same endpoint routing system as ASP.NET Core 3.0.

In this section I cover

- How endpoint routing works in ASP.NET Core
- The two types of routing available: convention-based routing and attribute routing
- How routing works for Razor Pages

At the end of this section you should have a good sense of how routing in ASP.NET Core works with Razor Pages.

5.2.1 Using endpoint routing in ASP.NET Core

Endpoint routing is fundamental to all but the simplest ASP.NET Core apps. It's implemented using two pieces of middleware, which you've seen previously:

- `EndpointMiddleware`—You use this middleware to *register* the endpoints in routing the system when you start your application. The middleware *executes* one of the endpoints at runtime.
- `EndpointRoutingMiddleware`—This middleware chooses *which* of the endpoints registered by the `EndpointMiddleware` should execute for a given request at runtime. To make it easier to distinguish between the two types of middleware, I'll be referring to this middleware as the `RoutingMiddleware` throughout this book.

The `EndpointMiddleware` is where you configure all the *endpoints* in your system. This is where you register your Razor Pages and MVC controllers, but you can also register additional handlers that fall outside of the MVC framework, such as health-check endpoints that confirm your application is still running.

> **DEFINITION** An *endpoint* in ASP.NET Core is some handler that returns a response. Each endpoint is associated with a URL pattern. Razor Page handlers and MVC controller action methods typically make up the bulk of the endpoints in an application, but you can also use simple middleware as an endpoint, or a health-check endpoint.

To register endpoints in your application, call `UseEndpoints` in the `Configure` method of Startup.cs. This method takes a configuration lambda action that defines the endpoints in your application, as shown in the following listing. You can automatically register all the Razor Pages in your application using extensions such as `MapRazorPages`. Additionally, you can register other endpoints explicitly using methods such as `MapGet`.

Listing 5.1 Registering multiple endpoints in `Startup.Configure`

```
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}
```

Add the EndpointRoutingMiddleware to the middleware pipeline.

Add the EndpointMiddleware to the pipeline and provide a configuration lambda.

Register a health-check endpoint at the route /healthz.

Register all the Razor Pages in your application as endpoints.

```
        endpoints.MapGet("/test", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Register an endpoint inline that returns "Hello World!" at the route /test.

Each endpoint is associated with a *route template* that defines which URLs the endpoint should match. You can see two route templates, `"/healthz"` and `"/test"`, in the previous listing.

> **DEFINITION** A *route template* is a URL pattern that is used to match against request URLs. They're strings of fixed values, like `"/test"` in the previous listing. They can also contain placeholders for variables, as you'll see in section 5.3.

The `EndpointMiddleware` stores the registered routes and endpoints in a dictionary, which it shares with the `RoutingMiddleware`. At runtime the `RoutingMiddleware` compares an incoming request to the routes registered in the dictionary. If the `RoutingMiddleware` finds a matching endpoint, it makes a note of which endpoint was selected and attaches that to the request's `HttpContext` object. It then calls the next middleware in the pipeline. When the request reaches the `EndpointMiddleware`, the middleware checks to see which endpoint was selected and executes it, as shown in figure 5.3.

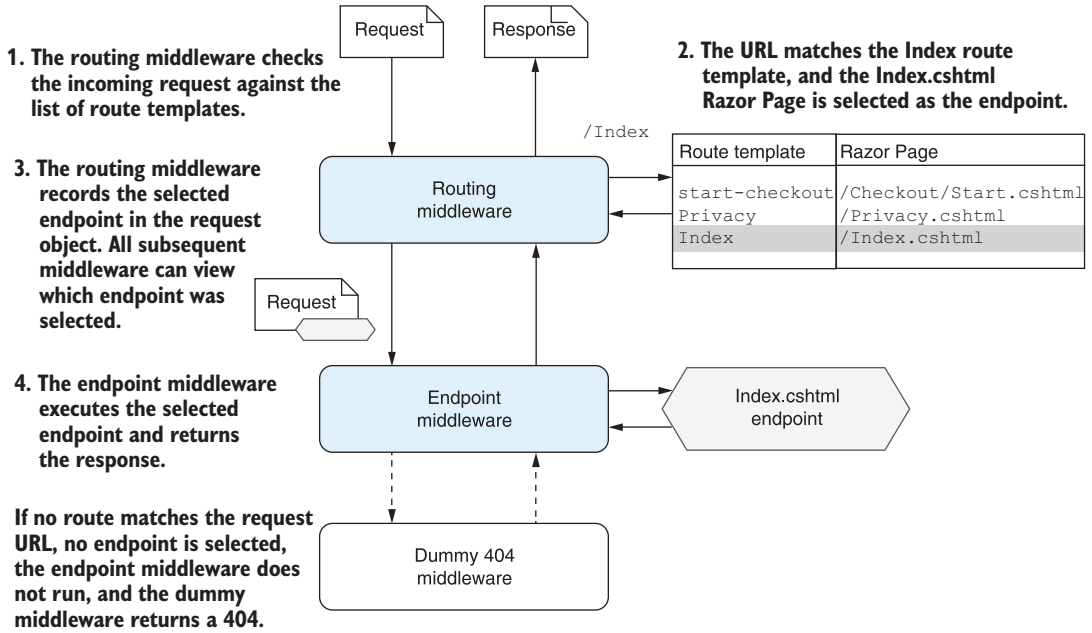

Figure 5.3 Endpoint routing uses a two-step process. The `RoutingMiddleware` selects which endpoint to execute, and the `EndpointMiddleware` executes it. If the request URL doesn't match a route template, the endpoint middleware will not generate a response.

If the request URL *doesn't* match a route template, the `RoutingMiddleware` doesn't select an endpoint, but the request still continues down the middleware pipeline. As no endpoint is selected, the `EndpointMiddleware` silently ignores the request and passes it to the next middleware in the pipeline. The `EndpointMiddleware` is typically the final middleware in the pipeline, so the "next" middleware is normally the dummy middleware that always returns a `404 Not Found` response, as you saw in chapter 3.

> **TIP** If the request URL does not match a route template, no endpoint is selected or executed. The whole middleware pipeline is still executed, but typically a 404 response is returned when the request reaches the dummy 404 middleware.

The advantage of having two separate pieces of middleware to handle this process might not be obvious at first blush. Figure 5.3 hinted at the main benefit—all middleware placed after the `RoutingMiddleware` can see which endpoint is *going* to be executed before it is.

> **NOTE** Only middleware placed *after* the `RoutingMiddleware` can detect which endpoint is going to be executed.

Figure 5.4 shows a more realistic middleware pipeline, where middleware is placed both *before* the `RoutingMiddleware` and *between* the `RoutingMiddleware` and the `EndpointMiddleware`.

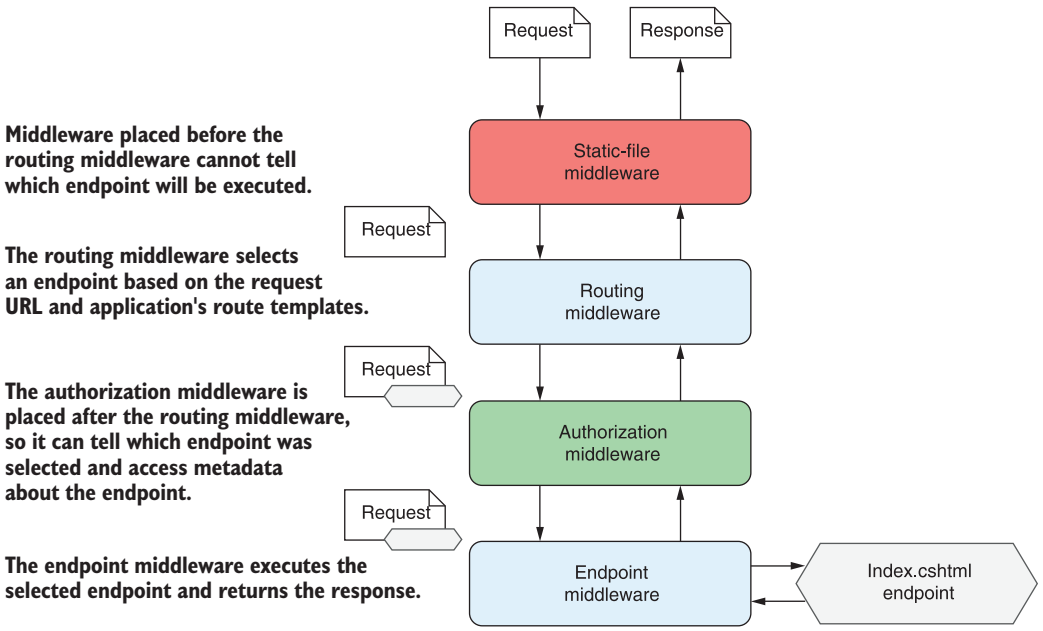

Figure 5.4 Middleware placed before the routing middleware doesn't know which endpoint the routing middleware will select. Middleware placed between the routing middleware and the endpoint middleware can see the selected endpoint.

The `StaticFileMiddleware` in figure 5.4 is placed *before* the `RoutingMiddleware`, so it executes before an endpoint is selected. Conversely, the `AuthorizationMiddleware` is placed *after* the `RoutingMiddleware`, so it can tell that the Index.cshtml Razor Page endpoint will be executed eventually. In addition, it can access certain metadata about the endpoint, such as its name and what the required permissions are to access the Razor Page.

> **TIP** The `AuthorizationMiddleware` needs to know which endpoint will be executed, so it must be placed *after* the `RoutingMiddleware` and *before* the `EndpointMiddleware` in your middleware pipeline. I discuss authorization in more detail in chapter 15.

It's important to remember the different roles of the two types of routing middleware when building your application. If you have a piece of middleware that needs to know which endpoint (if any) a given request will execute, then you need to make sure you place it after the `RoutingMiddleware` and before the `EndpointMiddleware`.

We've covered how the `RoutingMiddleware` and `EndpointMiddleware` interact to provide routing capabilities in ASP.NET Core, but we haven't yet looked at *how* the `RoutingMiddleware` matches the request URL to an endpoint. In the next section we'll look at the two different approaches used in ASP.NET Core.

5.2.2 Convention-based routing vs. attribute routing

Routing is a key part of ASP.NET Core, as it maps the incoming request's URL to a specific endpoint to execute. You have two different ways to define these URL-endpoint mappings in your application:

- Using global, convention-based routing
- Using attribute routing

Which approach you use will typically depend on whether you're using Razor Pages or MVC controllers, and whether you're building an API or a website (using HTML). These days I lean heavily toward attribute routing, as you'll see shortly.

Convention-based routing is defined globally for your application. You can use convention-based routes to map endpoints (MVC controller actions) in your application to URLs, but your MVC controllers must adhere strictly to the conventions you define. Traditionally, applications using MVC controllers to generate HTML tend to use this approach to routing. The downside to this approach is that it makes customizing the URLs for a subset of controllers and actions more difficult.

Alternatively, you can use attribute-based routes to tie a given URL to a specific endpoint. For MVC controllers, this involves placing `[Route]` attributes on the action methods themselves, hence the term *attribute-routing*. This provides a lot more flexibility, as you can explicitly define what the URL for each action method should be. This approach is generally more verbose than the convention-based approach, as it requires applying attributes to *every* action method in your application. Despite this, the additional flexibility it provides can be very useful, especially when building Web APIs.

Somewhat confusingly, Razor Pages uses *conventions* to generate *attribute routes*! In many ways this combination gives the best of both worlds—you get the predictability and terseness of convention-based routing with the easy customization of attribute routing. There are trade-offs to each of the approaches, as shown in table 5.1.

Table 5.1 The advantages and disadvantages of the routing styles available in ASP.NET Core

Routing style	Typical use	Advantages	Disadvantages
Convention-based routes	HTML-generating MVC controllers	Very terse definition in one location in your application Forces a consistent layout of MVC controllers	Routes are defined in a different place from your controllers. Overriding the route conventions can be tricky and error prone. Adds an extra layer of indirection when routing a request
Attribute routes	Web API MVC controllers	Gives complete control over route templates for every action. Routes are defined next to the action they execute.	Verbose compared to convention-based routing Can be easy to over-customize route templates. Route templates are scattered throughout your application, rather than in one location.
Convention-based generation of attribute routes	Razor Pages	Encourages consistent set of exposed URLs Terse when you stick to the conventions Easily override the route template for a single page. Customize conventions globally to change exposed URLs.	Possible to over-customize route templates You must calculate what the route template for a page is, rather than it being explicitly defined in your app.

So which approach should you use? My opinion is that convention-based routing is not worth the effort in 99% of cases and that you should stick to attribute routing. If you're following my advice of using Razor Pages, then you're already using attribute routing under the covers. Also, if you're creating APIs using MVC controllers, attribute routing is the best option and is the recommended approach.[2]

The only scenario where convention-based routing is used traditionally is if you're using MVC controllers to generate HTML. But if you are following my advice from chapter 4, you'll be using Razor Pages for HTML-generating applications, and only falling back to MVC controllers when completely necessary. For consistency, I would still stick with attribute routing in that scenario.

[2] .NET 5.0 also includes support for building simple APIs without the overhead (or convenience) of either MVC or Razor Pages. Filip W. has an excellent post on creating small, simple API apps using C# 9 and .NET 5.0: http://mng.bz/yYey.

> **NOTE** For the reasons above, this book focuses on attribute routing. Virtually all the features described in this section also apply to convention-based routing. For details on convention-based routing, see Microsoft's "Routing to controller actions in ASP.NET Core" documentation: http://mng.bz/ZP0O.

Whichever technique you use, you'll define your application's expected URLs using *route templates*. These define the pattern of the URL you're expecting, with placeholders for the parts that may vary.

> **DEFINITION** *Route templates* define the structure of known URLs in your application. They're strings with placeholders for variables that can contain optional values.

A single route template can match many different URLs. For example, the `/customer/1` and `/customer/2` URLs would both be matched by the `"customer/{id}"` route template. The route template syntax is powerful and contains many different features that are controlled by splitting a URL into multiple *segments*.

> **DEFINITION** A *segment* is a small contiguous section of a URL. It's separated from other URL segments by at least one character, often by the / character. Routing involves matching the segments of a URL to a route template.

For each route template, you can define

- Specific, expected strings
- Variable segments of the URL
- Optional segments of a URL
- Default values when an optional segment isn't provided
- Constraints on segments of a URL, such as ensuring that it's numeric

Most applications will use a variety of these features, but you'll often only use one or two features here and there. For the most part, the default convention-based attribute route templates generated by Razor Pages will be all you need. In the next section we'll look at those conventions and how routing maps a request's URL to a Razor Page in detail.

5.2.3 Routing to Razor Pages

As I mentioned in section 5.2.2, Razor Pages uses attribute routing by creating route templates based on conventions. ASP.NET Core creates a route template for every Razor Page in your app during app startup, when you call `MapRazorPages` in the `Configure` method of Startup.cs:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
```

For every Razor Page in your application, the framework uses the path of the Razor Page file relative to the Razor Pages root directory (`Pages/`), excluding the file extension (.cshtml). For example, if you have a Razor Page located at the path `Pages/Products/View.cshtml`, the framework creates a route template with the value `"Products/View"`, as shown in figure 5.5.

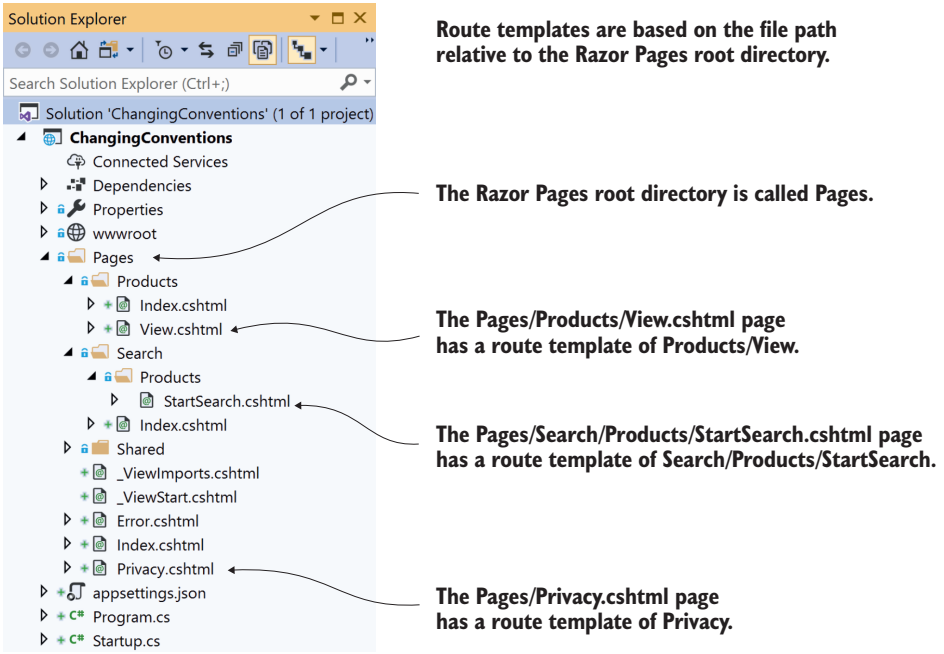

Figure 5.5 By default, route templates are generated for Razor Pages based on the path of the file relative to the root directory, Pages.

Requests to the URL `/products/view` match the route template `"Products/View"`, which in turn corresponds to the View.cshtml Razor Page. The `RoutingMiddleware` selects the View.cshtml Razor Page as the endpoint for the request, and the `EndpointMiddleware` executes the page's handler once the request reaches it in the middleware pipeline.

> **TIP** Routing is not case sensitive, so the request URL does not need to have the same URL casing as the route template to match.

In chapter 4 you learned that Razor Page handlers are the methods that are invoked on a Razor Page. When we say "a Razor Page is executed," we really mean "an instance of the Razor Page's `PageModel` is created, and a page handler on the model is invoked." Razor Pages can have multiple page handlers, so once the `RoutingMiddleware` selects

a Razor Page, the `EndpointMiddleware` still needs to choose which handler to execute. You'll learn how the framework selects which page handler to invoke in section 5.6.

By default, each Razor Page creates a single route template based on its file path. The exception to this rule is for Razor Pages that are called Index.cshtml. Index.cshtml pages create *two* route templates, one ending with `"Index"` and the other without an ending. For example, if you have a Razor Page at the path `Pages/ToDo/Index.cshtml`, that will generate two route templates:

- `"ToDo"`
- `"ToDo/Index"`

When either of these routes are matched, the same Index.cshtml Razor Page is selected. For example, if your application is running at the URL https://example.org, you can view the page by executing https://example.org/ToDo or https://example.org/ToDo/Index.

As a final example, consider the Razor Pages created by default when you create a Razor Pages application using Visual Studio or by running `dotnet new web` using the .NET CLI, as we did in chapter 2. The standard template includes three Razor Pages in the Pages directory:

- `Pages/Error.cshtml`
- `Pages/Index.cshtml`
- `Pages/Privacy.cshtml`

That creates a collection of four routes for the application, defined by the following templates:

- `""` maps to Index.cshtml
- `"Index"` maps to Index.cshtml
- `"Error"` maps to Error.cshtml
- `"Privacy"` maps to Privacy.cshtml

At this point, routing probably feels laughably trivial, but this is just the basics that you get for free with the default Razor Pages conventions, which is often sufficient for a large portion of any website. At some point, though, you'll find you need something more dynamic, such as an e-commerce site where you want each product to have its own URL, but which map to a single Razor Page. This is where route templates and route data come in and show the real power of routing.

5.3 Customizing Razor Page route templates

The route templates for a Razor Page are based on the file path by default, but you're also able to customize the final template for each page, or even to replace it entirely. In this section I show how to customize the route templates for individual pages, so you can customize your application's URLs and map multiple URLs to a single Razor Page.

Route templates have a rich, flexible syntax, but a simple example is shown in figure 5.6.

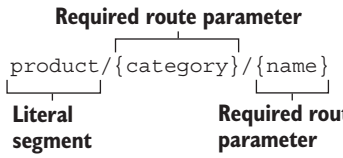

Figure 5.6 A simple route template showing a literal segment and two required route parameters

The routing middleware parses a route template by splitting it into a number of *segments*. A segment is typically separated by the / character, but it can be any valid character. Each segment is either

- *A literal value*—For example, `product` in figure 5.6
- *A route parameter*—For example, `{category}` and `{name}` in figure 5.6

Literal values must be matched exactly (ignoring case) by the request URL. If you need to match a particular URL exactly, you can use a template consisting only of literals. This is the default case for Razor Pages, as you saw in section 5.2.3; each Razor Page consists of a series of literal segments, for example `"ToDo/Index"`.

TIP Literal segments in ASP.NET Core are not case-sensitive.

Imagine you have a contact page in your application at the path `Pages/About/Contact.cshtml`. The route template for this page is `"About/Contact"`. This route template consists of only literal values and so only matches the exact URL. None of the following URLs would match this route template:

- `/about`
- `/about-us/contact`
- `/about/contact/email`
- `/about/contact-us`

Route parameters are sections of a URL that may vary but will still be a match for the template. They are defined by giving them a name and placing them in braces, such as `{category}` or `{name}`. When used in this way, the parameters are required, so there must be a segment in the request URL that they correspond to, but the value can vary.

TIP Some words can't be used as names for route parameters: `area`, `action`, `controller`, `handler`, and `page`.

The ability to use route parameters gives you great flexibility. For example, the simple route template `"{category}/{name}"` could be used to match all the product-page URLs in an e-commerce application, such as

- `/bags/rucksack-a`—Where `category=bags` and `name=rucksack-a`
- `/shoes/black-size9`—Where `category=shoes` and `name=black-size9`

But note that this template would *not* map the following URLs:

- `/socks/`—No `name` parameter specified
- `/trousers/mens/formal`—Extra URL segment, `formal`, not found in route template

When a route template defines a route parameter, and the route matches a URL, the value associated with the parameter is captured and stored in a dictionary of values associated with the request. These *route values* typically drive other behavior in the Razor Page, such as model binding.

> **DEFINITION** *Route values* are the values extracted from a URL based on a given route template. Each route parameter in a template will have an associated route value and they are stored as a string pair in a dictionary. They can be used during model binding, as you'll see in chapter 6.

Literal segments and route parameters are the two cornerstones of ASP.NET Core route templates. With these two concepts it's possible to build all manner of URLs for your application. But how can you customize a Razor Page to use one of these patterns?

5.3.1 Adding a segment to a Razor Page route template

To customize the Razor Page route template, you update the `@page` directive at the top of the Razor Page's .cshtml file. This directive must be the first thing in the Razor Page file for the page to be registered correctly.

> **NOTE** You must include the `@page` directive at the top of a Razor Page's .cshtml file. Without it, ASP.NET Core will not treat the file as a Razor Page, and you will not be able to view the page.

To add an additional segment to a Razor Page's route template, add a space followed by the desired route template, after the `@page` statement. For example, to add `"Extra"` to a Razor Page's route template, use

```
@page "Extra"
```

This *appends* the provided route template to the default template generated for the Razor Page. For example, the default route template for the Razor Page at `Pages/Privacy.html` is `"Privacy"`. With the preceding directive, the new route template for the page would be `"Privacy/Extra"`.

> **NOTE** The route template provided in the `@page` directive is *appended* to the end of the default template for the Razor Page.

The most common reason for customizing a Razor Page's route template like this is to add a *route parameter*. For example, you could have a single Razor Page for displaying the products in an e-commerce site at the path `Pages/Products.cshtml`, and use a route parameter in the `@page` directive

```
@page "{category}/{name}"
```

This would give a final route template of `Products/{category}/{name}`, which would match all of the following URLs:

- `/products/bags/white-rucksack`
- `/products/shoes/black-size9`
- `/Products/phones/iPhoneX`

It's very common to *add* route segments to the Razor Page template like this, but what if that's not enough? Maybe you don't want to have the `/products` segment at the start of the preceding URLs, or you want to use a completely custom URL for a page. Luckily that's just as easy to achieve.

5.3.2 Replacing a Razor Page route template completely

You'll be most productive working with Razor Pages if you can stick to the default routing conventions where possible, adding additional segments for route parameters where necessary. But sometimes you just need more control. That's often the case for important pages in your application, such as the checkout page for an e-commerce application, or even the product pages, as you saw in the previous section.

To specify a custom route for a Razor Page, prefix the route with `/` in the `@page` directive. For example, to remove the `"product/"` prefix from the route templates in the previous section, use this directive:

```
@page "/{category}/{name}"
```

Note that this directive includes the `"/"` at the start of the route, indicating that this is a *custom* route template, instead of an *addition*. The route template for this page will be `"{category}/{name}"`, no matter which Razor Page it is applied to.

Similarly, you can create a *static* custom template for a page by starting the template with a `"/"` and using only literal segments. For example,

```
@page "/checkout"
```

Wherever you place your checkout Razor Page within the Pages folder, using this directive ensures it always has the route template `"checkout"`, so it will always match the request URL `/checkout`.

> **TIP** You can also think of custom route templates that start with `"/"` as `absolute` route templates, while other route templates are `relative` to their location in the file hierarchy.

It's important to note that when you customize the route template for a Razor Page, both when appending to the default and when replacing it with a custom route, the *default template is no longer valid*. For example, if you use the `"checkout"` route template above on a Razor Page located at `Pages/Payment.cshtml`, you can *only* access it using the URL `/checkout`; the URL `/Payment` is no longer valid and will not execute the Razor Page.

TIP Customizing the route template for a Razor Page using the `@page` directive *replaces* the default route template for the page. In section 5.7 I show how you can add additional routes while preserving the default route template.

In this section you learned how to customize the route template for a Razor Page. In the next section we'll look in more depth at the route template syntax and some of the other features available.

5.4 Exploring the route template syntax

In addition to the basic elements of literals and route parameter segments, route templates have extra features that give you more control over your application's URLs. These features let you have optional URL segments, provide default values when a segment isn't specified, or place additional constraints on the values that are valid for a given route parameter. This section takes a look at these features and the ways you can apply them.

5.4.1 Using optional and default values

In the previous section, you saw a simple route template with a literal segment and two required routing parameters. In figure 5.7, you can see a more complex route that uses a number of additional features.

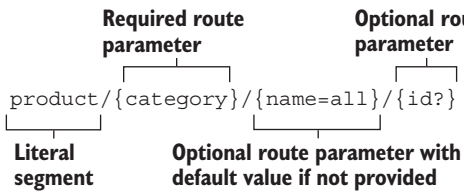

Figure 5.7 A more complex route template showing literal segments, named route parameters, optional parameters, and default values

The literal `product` segment and the required `{category}` parameter are the same as you saw in figure 5.6. The `{name}` parameter looks *similar*, but it has a default value specified for it using `=all`. If the URL doesn't contain a segment corresponding to the `{name}` parameter, the router will use the `all` value instead.

The final segment of figure 5.7, `{id?}`, defines an optional route parameter called `id`. This segment of the URL is optional—if present, the router will capture the value for the `{id}` parameter; if it isn't there, then it won't create a route value for `id`.

You can specify any number of route parameters in your templates, and these values will be available to you when it comes to model binding. The complex route template of figure 5.7 allows you to match a greater variety of URLs by making `{name}` and `{id}` optional, and providing a default for `{name}`. Table 5.2 shows some of the possible URLs this template would match and the corresponding route values the router would set.

Table 5.2 URLs that would match the template of figure 5.7 and their corresponding route values

URL	Route values
/product/shoes/formal/3	category=shoes, name=formal, id=3
/product/shoes/formal	category=shoes, name=formal
/product/shoes	category=shoes, name=all
/product/bags/satchels	category=bags, name=satchels
/product/phones	category=phones, name=all
/product/computers/laptops/ABC-123	category=computes, name=laptops, id=ABC-123

Note that there's no way to specify a value for the optional {id} parameter without also specifying the {category} and {name} parameters. You can *only* put an optional parameter (that doesn't have a default) at the *end* of a route template. For example, imagine your route template had an optional {category} parameter:

```
{category?}/{name}
```

Now try to think of a URL that would specify the {name} parameter, but not the {category}. It can't be done! Patterns like this won't cause errors per se; the category parameter is just essentially required, even though you've marked it as optional.

Using default values allows you to have multiple ways to call the same URL, which may be desirable in some cases. For example, given the route template in figure 5.7, the following two URLs are equivalent:

- /product/shoes
- /product/shoes/all

Both of these URLs will execute the same Razor Page, with the same route values of category=shoes and name=all. Using default values allows you to use shorter and more memorable URLs in your application for common URLs, but still have the flexibility to match a variety of routes in a single template.

5.4.2 Adding additional constraints to route parameters

By defining whether a route parameter is required or optional, and whether it has a default value, you can match a broad range of URLs with a pretty terse template syntax. Unfortunately, in some cases this can end up being a little *too* broad. Routing only matches URL segments to route parameters; it doesn't know anything about the data that you're expecting those route parameters to contain. If you consider a template similar to that in figure 5.7, "{category}/{name=all}/{id?}, the following URLs would all match:

- /shoes/sneakers/test
- /shoes/sneakers/123

- `/Account/ChangePassword`
- `/ShoppingCart/Checkout/Start`
- `/1/2/3`

These URLs are all perfectly valid given the template's syntax, but some might cause problems for your application. These URLs all have two or three segments, so the router happily assigns route values and matches the template when you might not want it to! These are the route values assigned:

- `/shoes/sneakers/test—category=shoes, name=sneakers, id=test`
- `/shoes/sneakers/123—category=shoes, name=sneakers, id=123`
- `/Account/ChangePassword—category=Account, name=ChangePassword`
- `/Cart/Checkout/Start—category=Cart, name=Checkout, id=Start`
- `/1/2/3—category=1, name=2, id=3`

Typically, the router passes route values to Razor Pages through a process called model binding, which you saw briefly in chapter 4 (and which we'll discuss in detail in the next chapter). For example, a Razor Page with the handler `public void OnGet(int id)` would obtain the `id` argument from the `id` route value. If the `id` route parameter ends up assigned a *non-integer* value from the URL, you'll get an exception when it's bound to the *integer* `id` parameter.

To avoid this problem, it's possible to add additional *constraints* to a route template that must be satisfied for a URL to be considered a match. Constraints can be defined in a route template for a given route parameter using : (a colon). For example, `{id:int}` would add the `IntRouteConstraint` to the `id` parameter. For a given URL to be considered a match, the value assigned to the `id` route value must be convertible to an integer.

You can apply a large number of route constraints to route templates to ensure that route values are convertible to appropriate types. You can also check more advanced constraints, for example, that an integer value has a particular minimum value, or that a string value has a maximum length. Table 5.3 describes a number of the possible constraints available, but you can find a more complete list online in Microsoft's "Routing in ASP.NET Core" documentation at http://mng.bz/xmae.

Table 5.3 A few route constraints and their behavior when applied

Constraint	Example	Match examples	Description
`int`	`{qty:int}`	`123, -123, 0`	Matches any `integer`
`Guid`	`{id:guid}`	`d071b70c-a812-4b54-87d2-7769528e2814`	Matches any `Guid`
`decimal`	`{cost:decimal}`	`29.99, 52, -1.01`	Matches any `decimal` value
`min(value)`	`{age:min(18)}`	`18, 20`	Matches `integer` values of 18 or greater

Table 5.3 A few route constraints and their behavior when applied *(continued)*

Constraint	Example	Match examples	Description
`length(value)`	`{name:length(6)}`	`andrew,123456`	Matches `string` values with a length of 6
`optional int`	`{qty:int?}`	`123, -123, 0, null`	Optionally matches any `integer`
`optional int max(value)`	`{qty:int:max(10)?}`	`3, -123, 0, null`	Optionally matches any `integer` of 10 or less

TIP As you can see from table 5.3, you can also combine multiple constraints by separating the constraints with colons.

Using constraints allows you to narrow down the URLs that a given route template will match. When the routing middleware matches a URL to a route template, it interrogates the constraints to check that they're all valid. If they aren't valid, the route template isn't considered a match, and the Razor Page won't be executed.

WARNING Don't use route constraints to validate general input, such as to check that an email address is valid. Doing so will result in 404 "Page not found" errors, which will be confusing for the user.

Constraints are best used sparingly, but they can be useful when you have strict requirements on the URLs used by the application, as they can allow you to work around some otherwise tricky combinations.

Constraints and ordering in attribute routing

If you have a well-designed set of URLs for your application, you will probably find that you don't really need to use route constraints. Route constraints really come in useful when you have "overlapping" route templates.

For example, imagine you have a Razor Page with the route template `"{number}/{name}"` and another with the template `"{product}/{id}"`. When a request with the URL `/shoes/123` arrives, which template is chosen? They *both* match, so the routing middleware panics and throws an exception. Not ideal.

Using conventions can fix this. For example, if you update the first template to `"{number:int}/{name}"`, then the integer constraint means the URL is no longer a match, and the routing middleware can choose correctly. Note, however, that the URL `/123/shoes` *does* still match both route templates, so you're not out of the woods.

Generally, you should avoid overlapping route templates like these, as they're often confusing and more trouble than they're worth.

Attribute routing (used by Razor Pages and MVC controllers for building APIs) allows you to explicitly control the *order* the routing middleware looks at your route templates, which can also be used to resolve issues like the one above. However, if you

(continued)

find yourself needing to manually control the order, this is a very strong indicator that your URLs are confusing.[a]

If your route templates are well defined, such that each URL only maps to a single template, ASP.NET Core routing will work without any difficulties. Sticking to the built-in conventions as far as possible is the best way to stay on the happy path!

[a] If you're really sure you do need to control route template ordering, see the documentation at http://mng.bz/MXqo. Note that you can only control the order for additional routes added using conventions, as you'll see in section 5.7.

We're coming to the end of our look at route templates, but before we move on there's one more type of parameter to think about: the catch-all parameter.

5.4.3 Matching arbitrary URLs with the catch-all parameter

You've already seen how route templates take URL segments and attempt to match them to parameters or literal strings. These segments normally split around the slash character, /, so the route parameters themselves won't contain a slash. What do you do if you need them to contain a slash, or you don't know how many segments you're going to have?

Imagine you are building a currency-converter application that shows the exchange rate from one currency to one or more other currencies. You're told that the URLs for this page should contain all the currencies as separate segments. Here are some examples:

- `/USD/convert/GBP`—Show USD with exchange rate to GBP
- `/USD/convert/GBP/EUR`—Show USD with exchange rates to GBP and EUR
- `/USD/convert/GBP/EUR/CAD`—Show USD with exchange rates for GBP, EUR, and CAD

If you want to support showing *any* number of currencies as the URLs above do, you need a way of capturing *everything* after the `convert` segment. You could achieve this for the `Pages/Convert.cshtml` Razor Page by using a catch-all parameter in the `@page` directive, as shown in figure 5.8.

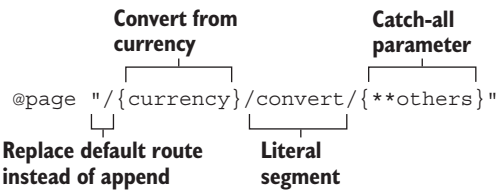

Figure 5.8 You can use catch-all parameters to match the remainder of a URL. Catch-all parameters may include the "/" character or may be an empty string.

Catch-all parameters can be declared using either one or two asterisks inside the parameter definition, like {*others} or {**others}. These will match the remaining unmatched portion of a URL, including any slashes or other characters that aren't part of earlier parameters. They can also match an empty string. For the USD/convert/GBP/EUR URL, the value of the route value others would be the single string "GBP/EUR".

> **TIP** Catch-all parameters are greedy and will capture the whole unmatched portion of a URL. Where possible, to avoid confusion, avoid defining route templates with catch-all parameters that overlap other route templates.

The one- and two-asterisk versions of the catch-all parameter behave identically when routing an incoming request to a Razor Page. The difference only comes when you're *generating* URLs (which we'll cover in the next section): the one-asterisk version URL encodes forward slashes, and the two-asterisk version doesn't. The round-trip behavior of the two-asterisk version is typically what you want.[3]

You read that correctly—mapping URLs to Razor Pages is only half of the responsibilities of the routing system in ASP.NET Core. It's also used to *generate* URLs so that you can easily reference your Razor Pages from other parts of your application.

5.5 Generating URLs from route parameters

In this section, we'll look at the other half of routing—generating URLs. You'll learn how to generate URLs as a string you can use in your code, and how to automatically send redirect URLs as a response from your Razor Pages.

One of the by-products of using the routing infrastructure in ASP.NET Core is that your URLs can be somewhat fluid. If you rename a Razor Page, the URL associated with that page will also change. For example, renaming the Pages/Cart.cshtml page to Pages/Basket/View.cshtml would cause the URL you use to access the page to change from /Cart to /Basket/View.

Trying to manually manage these links within your app would be a recipe for heartache, broken links, and 404s. If your URLs were hardcoded, you'd have to remember to do a find-and-replace with every rename!

Luckily, you can use the routing infrastructure to generate appropriate URLs dynamically at runtime instead, freeing you from the burden. Conceptually, this is almost an exact reverse of the process of mapping a URL to a Razor Page, as shown in figure 5.9. In the "routing" case, the routing middleware takes a URL, matches it to a route template, and splits it into route values. In the "URL generation" case, the generator takes in the *route values* and combines them with a route template to build a URL.

[3] For details and examples of this behavior, see the documentation: http://mng.bz/aoGo.

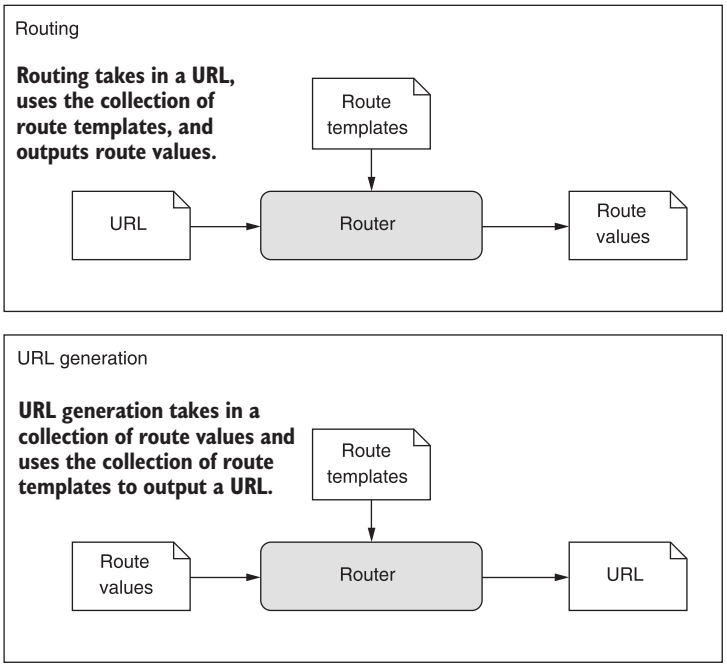

Figure 5.9 A comparison between routing and URL generation. Routing takes in a URL and generates route values, but URL generation uses route values to generate a URL.

5.5.1 Generating URLs for a Razor Page

You will need to generate URLs in various places in your application, and one common location is in your Razor Pages and MVC controllers. The following listing shows how you could generate a link to the `Pages/Currency/View.cshtml` Razor Page, using the `Url` helper from the `PageModel` base class.

Listing 5.2 Generating a URL using `IUrlHelper` and the Razor Page name

```
public class IndexModel : PageModel
{
    public void OnGet()
    {
        var url = Url.Page("Currency/View", new { code = "USD" });
    }
}
```

Deriving from PageModel gives access to the Url property.

You provide the relative path to the Razor Page, along with any additional route values.

The `Url` property is an instance of `IUrlHelper` that allows you to easily generate URLs for your application by referencing other Razor Pages by their file path. It exposes a `Page` method to which you pass the name of the Razor Page and any additional route data. The route data is packaged as key-value pairs into a single C# anonymous object.

If you need to pass more than one route value, you can add additional properties to the anonymous object. The helper will then generate a URL based on the referenced page's route template.

> **TIP** You can provide the *relative* file path to the Razor Page, as shown in listing 5.2. Alternatively, you can provide the *absolute* file path (relative to the Pages folder) by starting the path with a `"/"`, for example, `"/Currency/View"`.

The `IUrlHelper` has several different overloads of the `Page` method. Some of these methods allow you to specify a specific page handler, others let you generate an absolute URL instead of a relative URL, and some let you pass in additional route values.

In listing 5.2, as well as providing the file path, I passed in an anonymous object, `new { code = "USD" }`. This object provides additional route values when generating the URL, in this case setting the `code` parameter to `"USD"`.

If a selected route explicitly includes the defined route value in its definition, such as in the `"Currency/View/{code}"` route template, the route value will be used in the URL path, giving `/Currency/View/GBP`.

If a route doesn't contain the route value explicitly, as in the `"Currency/View"` template, the route value is appended as additional data as part of the query string, for example `/Currency/View?code=GBP`.

Generating URLs based on the page you want to execute is convenient, and it's the usual approach taken in most cases. If you're using MVC controllers for your APIs, the process is much the same as for Razor Pages, though the methods are slightly different.

5.5.2 Generating URLs for an MVC controller

Generating URLs for MVC controllers is very similar to Razor Pages. The main difference is that you use the `Action` method on the `IUrlHelper`, and you provide an MVC controller name and action name instead of a page path. The following listing shows an MVC controller generating a link from one action method to another, using the `Url` helper from the `Controller` base class.

Listing 5.3 Generating a URL using `IUrlHelper` and the action name

```
public class CurrencyController : Controller
{
    [HttpGet("currency/index")]
    public IActionResult Index()
    {
        var url = Url.Action("View", "Currency",
            new { code = "USD" });
        return Content($"The URL is {url}");
    }

    [HttpGet("currency/view/{code}")]
    public IActionResult View(string code)
    {
```

Deriving from Controller gives access to the Url property.

You provide the action and controller name, along with any additional route values.

This will return "The URL is /Currency/View/USD".

The URL generated will route to the View action method.

```
        /* method implementation*/
    }
}
```

You can call the `Action` and `Page` methods on `IUrlHelper` from both Razor Pages and MVC controllers, so you can generate links back and forth between them if you need to. The important question is, what is the *destination* of the URL? If the URL you need refers to a Razor Page, use the `Page` method. If the destination is an MVC action, use the `Action` method.

> **TIP** Instead of using strings for the name of the action method, use the C# 6 `nameof` operator to make the value refactor-safe, for example, `nameof(View)`.

If you're routing to an action in the same controller, you can use a different overload of `Action` that omits the controller name when generating the URL. The `IUrlHelper` uses *ambient values* from the current request and overrides these with any specific values you provide.

> **DEFINITION** Ambient values are the route values for the current request. They include `controller` and `action` when called from an MVC controller but can also include additional route values that were set when the action or Razor Page was initially located using routing. See Microsoft's "Routing in ASP.NET Core" documentation for further details: http://mng.bz/xmae.

Generating URLs using the `Url` property doesn't tend to be very common in practice. Instead, it's more common to generate URLs implicitly with an `ActionResult`.

5.5.3 Generating URLs with ActionResults

You've seen how to generate a string containing a URL for both Razor Pages and MVC actions. This is useful if you need to display the URL to a user, or to include the URL in an API response, for example. However, you don't need to display URLs very often. More commonly, you want to *automatically redirect* a user to a URL. For that situation you can use an `ActionResult` to handle the URL generation instead.

The following listing shows how you can generate a URL that automatically redirects a user to a different Razor Page using an `ActionResult`. The `RedirectToPage` method takes the path to a Razor Page and any required route parameters, and generates a URL in the same way as the `Url.Page` method. The framework automatically sends the generated URL as the response, so you never see the URL in your code. The user's browser then reads the URL from the response and automatically redirects to the new page.

Listing 5.4 Generating a redirect URL from an `ActionResult`

```
public class CurrencyModel : PageModel
{
    public IActionResult OnGetRedirectToPage()
    {
```

```
        return RedirectToPage("Currency/View", new { id = 5 });
    }
}
```

The RedirectToPage method generates a RedirectToPageResult with the generated URL.

You can use a similar method, `RedirectToAction`, to automatically redirect to an MVC action instead. Just as with the `Page` and `Action` methods, it is the *destination* that controls whether you need to use `RedirectToPage` or `RedirectToAction`. `RedirectToAction` is only necessary if you're using MVC controllers to generate HTML instead of Razor Pages.

> **TIP** I recommend you use Razor Pages instead of MVC controllers for HTML generation. For a discussion of the benefits of Razor Pages, refer to chapter 4.

As well as generating URLs from your Razor Pages and MVC controllers, you'll often find you need to generate URLs when building HTML in your views. This is necessary in order to provide navigation links in your web application. You'll see how to achieve this when we look at Razor Tag Helpers in chapter 8.

If you need to generate URLs from parts of your application outside of the Razor Page or MVC infrastructure, you won't be able to use the `IUrlHelper` helper or an `ActionResult`. Instead, you can use the `LinkGenerator` class.

5.5.4 Generating URLs from other parts of your application

If you're writing your Razor Pages and MVC controllers following the advice from chapter 4, you should be trying to keep your Razor Pages relatively simple. That requires you to execute your application's business and domain logic in separate classes and services.

For the most part, the URLs your application uses *shouldn't* be part of your domain logic. That makes it easier for your application to evolve over time, or even to change completely. For example, you may want to create a mobile application that reuses the business logic from an ASP.NET Core app. In that case, using URLs in the business logic wouldn't make sense, as they wouldn't be correct when the logic is called from the mobile app!

> **TIP** Where possible, try to keep knowledge of the frontend application design out of your business logic. This pattern is known generally as the Dependency Inversion principle.[4]

Unfortunately, sometimes that separation is not possible, or it makes things significantly more complicated. One example might be when you're creating emails in a background service—it's likely you'll need to include a link to your application in the email. The `LinkGenerator` class lets you generate that URL, so that it updates automatically if the routes in your application change.

---

[4] Steve Smith maintains a project template that applies this pattern rigorously, called the Clean Architecture. You can find the template, along with many useful links and books on the topic, here: https://github.com/ardalis/CleanArchitecture.

The `LinkGenerator` class is available in any part of your application, so you can use it inside middleware and any other services. You can use it from Razor Pages and MVC too, if you wish, but the `IUrlHelper` is typically easier and hides some details of using the `LinkGenerator`.

`LinkGenerator` has various methods for generating URLs, such as `GetPathByPage`, `GetPathByAction`, and `GetUriByPage`, as shown in the following listing. There are some subtleties to using these methods, especially in complex middleware pipelines, so stick to the `IUrlHelper` where possible, and be sure to consult the documentation if you have problems.[5]

Listing 5.5 Generating URLs using the `LinkGeneratorClass`

```
public class CurrencyModel : PageModel
{
    private readonly LinkGenerator _link;
    public CurrencyModel(LinkGenerator linkGenerator)
    {
        _link = linkGenerator;
    }

    public void OnGet ()
    {
        var url1 = Url.Page("Currency/View", new { id = 5 });
        var url3 = _link.GetPathByPage(
                        HttpContext,
                        "/Currency/View",
                        values: new { id = 5 });
        var url2 = _link.GetPathByPage(
                        "/Currency/View",
                        values: new { id = 5 });
        var url4 = _link.GetUriByPage(
                        page: "/Currency/View",
                        handler: null,
                        values: new { id = 5 },
                        scheme: "https",
                        host: new HostString("example.com"));
    }
}
```

LinkGenerator can be accessed using dependency injection.

Url can generate relative paths using Url.Page. You can use relative or absolute Page paths.

GetPathByPage is equivalent to Url.Page when you pass in HttpContext. You can't use relative paths.

Other overloads don't require an HttpContext.

GetUriByPage generates an absolute URL instead of a relative URL.

Whether you're generating URLs using the `IUrlHelper` or `LinkGenerator` you need to be careful when using the route generation methods. Make sure you provide the correct Razor Page path and any necessary route parameters. If you get something wrong—you have a typo in your path or you forgot to include a required route parameter, for example—the URL generated will be `null`. It's worth checking the generated URL for `null` explicitly, just to be sure there's no problems.

[5] You can find the `LinkGenerator` documentation here: http://mng.bz/goWx.

So far in this chapter we've looked extensively at how incoming requests are routed to Razor Pages, but we haven't really seen where page handlers come into it. In the next section we'll look at page handlers and how you can have multiple handlers on a Razor Page.

5.6 Selecting a page handler to invoke

At the start of this chapter I said routing was about mapping URLs to a *handler*. For Razor Pages, that means a *page handler*, but so far we've only been talking about routing based on a Razor Page's route template. In this section you'll learn how the `EndpointMiddleware` selects which page handler to invoke when it executes a Razor Page.

You learned about page handlers in chapter 4, and their role within Razor Pages, but we haven't discussed how a page handler is *selected* for a given request. Razor Pages can have multiple handlers, so if the `RoutingMiddleware` selects a Razor Page, the `EndpointMiddleware` still needs to know how to choose which handler to execute.

Consider the Razor Page `SearchModel` shown in the following listing. This Razor Page has three handlers: `OnGet`, `OnPostAsync`, and `OnPostCustomSearch`. The bodies of the handler methods aren't shown, as, at this point, we're only interested in how the `RoutingMiddleware` chooses which handler to invoke.

Listing 5.6 Razor Page with multiple page handlers

```
public class SearchModel : PageModel
{
    public void OnGet()                          <-- Handles GET requests
    {
        // Handler implementation
    }

    public Task OnPostAsync()                    <-- Handles POST requests. The async suffix is optional and is ignored for routing purposes.
    {
        // Handler implementation
    }

    public void OnPostCustomSearch()             <-- Handles POST requests where the handler route value has the value CustomSearch
    {
        // Handler implementation
    }
}
```

Razor Pages can contain any number of page handlers, but only one runs in response to a given request. When the `EndpointMiddleware` executes a selected Razor Page, it selects a page handler to invoke based on two variables:

- The HTTP verb used in the request (for example `GET`, `POST`, or `DELETE`)
- The value of the `handler` route value

The `handler` route value typically comes from a query string value in the request URL, for example, `/Search?handler=CustomSearch`. If you don't like the look of query strings (I don't!) you can include the `{handler}` route parameter in your Razor Page's

route template. For example, for the Search page in listing 5.6, you could update the page's directive to

```
@page "{handler}"
```

This would give a complete route template something like `"Search/{handler}"`, which would match URLs such as `/Search/CustomSearch`.

The `EndpointMiddleware` uses the handler route value and the HTTP verb together with a standard naming convention to identify which page handler to execute, as shown in figure 5.10. The handler parameter is optional and is typically provided as part of the request's query string or as a route parameter, as described above. The async suffix is also optional and is often used when the handler uses asynchronous programming constructs such as `Task` or `async/await`.[6]

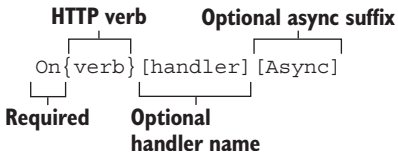

Figure 5.10 Razor Page handlers are matched to a request based on the HTTP verb and the optional handler parameter.

Based on this convention, we can now identify what type of request each page handler in listing 5.6 corresponds to:

- `OnGet`—Invoked for `GET` requests that don't specify a `handler` value.
- `OnPostAsync`—Invoked for `POST` requests that don't specify a `handler` value. Returns a `Task`, so it uses the `Async` suffix, which is ignored for routing purposes.
- `OnPostCustomSearch`—Invoked for `POST` requests that specify a `handler` value of `"CustomSearch"`.

The Razor Page in listing 5.6 specifies three handlers, so it can handle only three verb-handler pairs. But what happens if you get a request that doesn't match these, such as a request using the `DELETE` verb, a `GET` request with a non-blank `handler` value, or a `POST` request with an unrecognized `handler` value?

For all these cases, the `EndpointMiddleware` executes an *implicit* page handler instead. Implicit page handlers contain no logic; they just render the Razor view. For example, if you sent a `DELETE` request to the Razor Page in listing 5.6, an implicit handler would be executed. The implicit page handler is equivalent to the following handler code:

```
public void OnDelete() { }
```

[6] The async suffix naming convention is suggested by Microsoft, though it is unpopular with some developers. NServiceBus provides a reasoned argument against it here (along with Microsoft's advice): http://mng.bz/e59P.

DEFINITION If a page handler does not match a request's HTTP verb and handler value, an *implicit* page handler is executed that renders the associated Razor view. Implicit page handlers take part in model binding and use page filters but execute no logic.

There's one exception to the implicit page handler rule: if a request uses the `HEAD` verb, and there is no corresponding `OnHead` handler, Razor Pages will execute the `OnGet` handler instead (if it exists).[7]

At this point we've covered mapping request URLs to Razor Pages and generating URLs using the routing infrastructure, but most of the URLs we've been using have been kind of ugly. If seeing capital letters in your URLs bothers you, the next section is for you. In the next section we'll customize the conventions your application uses to generate route templates.

5.7 Customizing conventions with Razor Pages

Razor Pages is built on a series of conventions that are designed to reduce the amount of boilerplate code you need to write. In this section you'll see some of the ways you can customize those conventions. By customizing the conventions Razor Pages uses in your application, you get full control over your application's URLs without having to manually customize every Razor Page's route template.

By default, ASP.NET Core generates URLs that match the filenames of your Razor Pages very closely. For example, the Razor Page located at the path `Pages/Products/ProductDetails.cshtml`, would correspond to the route template `Products/Product-Details`.

These days, it's not very common to see capital letters in URLs. Similarly, words in URLs are usually separated using "kebab-case" rather than "PascalCase", for example, `product-details` instead of `ProductDetails`. Finally, it's also common to ensure your URLs always end with a trailing slash, for example, `/product-details/` instead of `/product-details`. Razor Pages gives you complete control over the conventions your application uses to generate route templates, but these are two common changes I make.

The following listing shows how you can ensure URLs are always lowercase and that they always have a trailing slash. You can change these conventions by configuring a `RouteOptions` object in Startup.cs. This object contains configuration for the whole ASP.NET Core routing infrastructure, so any changes you make will apply to both Razor Pages and MVC. You'll learn more about configuring options in chapter 10.

Listing 5.7 Configuring routing conventions using `RouteOptions` in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
```

Add the standard Razor Pages services.

[7] HEAD requests are typically sent automatically by the browser and don't return a response body. They're often used for security purposes, as you'll see in chapter 18.

Configure the RouteOptions object by providing a configuration method.

```
    services.Configure<RouteOptions>(options =>
    {
        options.AppendTrailingSlash = true;
        options.LowercaseUrls = true;
        options.LowercaseQueryStrings = true;
    });
}
```

You can change the conventions used to generate URLs. By default, these properties are false.

To use kebab-case for your application, annoyingly you must create a custom parameter transformer. This is a somewhat advanced topic, but it's relatively simple to implement in this case. The following listing shows how you can create a parameter transformer that uses a regular expression to replace PascalCase values in a generated URL with kebab-case.

Listing 5.8 Creating a kebab-case parameter transformer

```
public class KebabCaseParameterTransformer
    : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) return null;

        return Regex.Replace(value.ToString(),
            "([a-z])([A-Z])", "$1-$2").ToLower();
    }
}
```

Create a class that implements the parameter transformer interface.

Guard against null values to avoid runtime exceptions.

The regular expression replaces PascalCase patterns with kebab-case.

You can register the parameter transformer in your application with the `AddRazorPagesOptions` extension method in Startup.cs. This method is chained after the `AddRazorPages` method and can be used to completely customize the conventions used by Razor Pages. The following listing shows how to register the kebab-case transformer. It also shows how to add an extra page route convention for a given Razor Page.

Listing 5.9 Registering a parameter transformer using `RazorPagesOptions`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorPagesOptions(opts =>
        {
            opts.Conventions.Add(
                new PageRouteTransformerConvention(
                    new KebabCaseParameterTransformer()));
            opts.Conventions.AddPageRoute(
                "/Search/Products/StartSearch", "/search-products");
        });
}
```

AddRazorPagesOptions can be used to customize the conventions used by Razor Pages.

Registers the parameter transformer as a convention used by all Razor Pages

AddPageRoute adds an additional route template to Pages/Search/Products/StartSearch.cshtml.

The `AddPageRoute` convention adds an alternative way to execute a single Razor Page. Unlike when you customize the route template for a Razor Page using the `@page` directive, using `AddPageRoute` *adds an extra route template* to the page instead of replacing the default. That means there are two route templates that can access the page.

There are many other ways you can customize the conventions for your Razor Pages applications, but most of the time that's not necessary. If you do find you need to customize all the pages in your application in some way, such as to add an extra header to every page's response, you can use a custom convention. Microsoft's "Razor Pages route and app conventions in ASP.NET Core" documentation contains details on everything that's available: http://mng.bz/A0BK.

Conventions are a key feature of Razor Pages, and you should lean on them whenever you can. While you *can* manually override the route templates for individual Razor Pages, as you've seen in previous sections, I advise against it where possible. In particular,

- *Avoid* replacing the route template with an absolute path in a page's `@page` directive.
- *Avoid* adding literal segments to the `@page` directive. Rely on the file hierarchy instead.
- *Avoid* adding additional route templates to a Razor Page with the `AddPageRoute` convention. Having multiple ways to access a page can sometimes be confusing.
- *Do* add route parameters to the `@page` directive to make your routes dynamic, for example, `@page {name}`.
- *Do* consider using global conventions when you want to change the route templates for all your Razor Pages, such as using kebab-case, as you saw in the previous section.

In a nutshell, these rules amount to "stick to the conventions." The danger, if you don't, is that you may accidentally create two Razor Pages that have overlapping route templates. Unfortunately, if you end up in that situation, you won't get an error at compile time. Instead, you'll get an exception at runtime when your application receives a request that matches multiple route templates, as shown in figure 5.11.

Congratulations, you've made it all the way through this detailed discussion on routing! Routing is one of those topics that people often get stuck on when they come to building an application, which can be frustrating. We'll revisit routing again when I describe how to create Web APIs in chapter 9, but rest assured, you've already covered all the tricky details in this chapter!

In chapter 6 we'll dive into model binding. You'll see how the route values generated during routing are bound to your action method (MVC) or page handler (Razor Pages) parameters, and perhaps more importantly, how to validate the values you're provided.

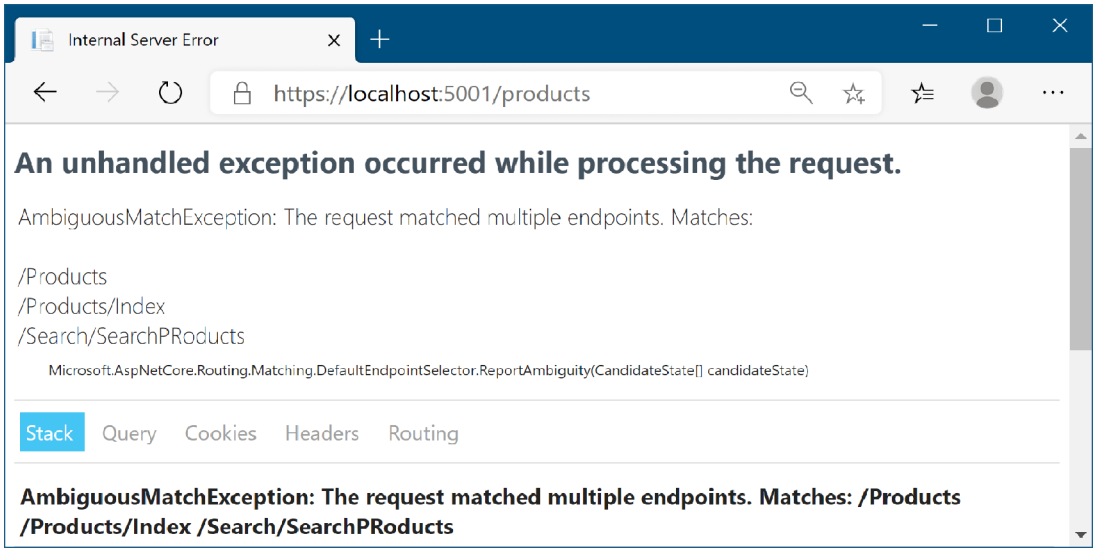

Figure 5.11 If multiple Razor Pages are registered with overlapping route templates, you'll get an exception at runtime when the router can't work out which one to select.

Summary

- Routing is the process of mapping an incoming request URL to a Razor Page that will execute to generate a response. You can use routing to decouple your URLs from the files in your project and to have multiple URLs map to the same Razor Page.
- ASP.NET Core uses two pieces of middleware for routing. The `EndpointRoutingMiddleware` is added in Startup.cs by calling `UseRouting()` and the `EndpointMiddleware` is added by calling `UseEndpoints()`.
- The `EndpointRoutingMiddleware` selects which endpoint should be executed by using routing to match the request URL. The `EndpointMiddleware` executes the endpoint.
- Any middleware placed between the calls to `UseRouting()` and `UseEndpoints()` can tell which endpoint will be executed for the request.
- Route templates define the structure of known URLs in your application. They're strings with placeholders for variables that can contain optional values and map to Razor Pages or to MVC controller actions.
- Route parameters are variable values extracted from a request's URL.
- Route parameters can be optional and can use default values when a value is missing.
- Route parameters can have constraints that restrict the possible values allowed. If a route parameter doesn't match its constraints, the route isn't considered a match.

- Don't use route constraints as general input validators. Use them to disambiguate between two similar routes.
- Use a catch-all parameter to capture the remainder of a URL into a route value.
- You can use the routing infrastructure to generate internal URLs for your application.
- The `IUrlHelper` can be used to generate URLs as a string based on an action name or Razor Page.
- You can use the `RedirectToAction` and `RedirectToPage` methods to generate URLs while also generating a redirect response.
- The `LinkGenerator` can be used to generate URLs from other services in your application, where you don't have access to an `HttpContext` object.
- When a Razor Page is executed, a single page handler is invoked based on the HTTP verb of the request and the value of the `handler` route value.
- If there is no page handler for a request, an implicit page handler is used that renders the Razor view.
- You can control the routing conventions used by ASP.NET Core by configuring the `RouteOptions` object, such as to force all URLs to be lowercase, or to always append a trailing slash.
- You can add additional routing conventions for Razor Pages by calling `AddRazorPagesOptions()` after `AddRazorPages()` in Startup.cs. These conventions can control how route parameters are displayed or can add additional route templates for specific Razor Pages.
- Where possible, avoid customizing the route templates for a Razor Page and rely on the conventions instead.